The Eldon 2 operating system for KDF9

M. Wells, D. Holdsworth, and A. P. McCann
University of Leeds

Eldon 2* is an operating system for a KDF9 computer which has been developed by the Computing Laboratory at Leeds University. The system provides conversational file maintenance, multiple remote job entry into either a background or foreground job queue (the latter offering a turn round time of a few minutes), and efficient processing of background jobs. A disc based filing system with archive and retrieval facilities for less frequently used files is accessible via a variety of external media and the system incorporates automatic accounting for all work processed. Use of this system has increased CPU utilisation from about 50% to about 85% and allows the processing of over 1,000 jobs per day.

(Received December 1969)

The 'Flowers' upgrading of University KDF9's during the period 1966–8 provided enhanced facilities on a number of machines, including that at Leeds. In parallel with the provision of extra hardware an *ad hoc* committee, under the chairmanship of Dr. K. W. Morton, was set up to make recommendations for the provision of suitable software for these upgraded machines. This committee proposed improvements and extensions of the Egdon operating system (Burns, Hawkins, Judd, and Venn, 1966). At the same time, English Electric Co. had completed preliminary work on a disc-based system 'Prompt', and we at Leeds were encouraged to carry on and develop this approach.

Our aims in working on the system have been two-fold. One, rather abstract, was to discover as much as possible about the design and construction of an operating system, so as to teach the subject better. The second, more concrete, was to provide an efficient and reliable operating system for a medium size machine, so as to use it better.

The system that we conceived was based heavily on the disc file, and would allow users to construct and edit files of information via a variety of external media. Users would be able to translate a program, recover diagnostic messages and to run the program which would find its data on the disc and return its output either to the disc or directly to an external medium. All of this was to be achieved while occupying a minimum of core store and allowing an unimpeded flow of operator initiated jobs. The whole system was to allow the collection of statistical data, both as a means of controlling resources, and of locating potential bottlenecks.

The system was not required to provide conversational interaction with a program while it was running; it was not required to provide protection against malicious (as distinct from accidental) attempts to access files not belonging to a user; it was not required to cope with very large programs, written in a number of different source languages. We had good reasons for this deliberate curtailment of our aims. Lacking a drum, and with only 32K of core, KDF9 is simply not adequately equipped for job-swapping. We had run a file-maintenance scheme for a number of years, and had not the slightest evidence to suggest that anyone had ever deliberately tampered with other users' files. The compilers available, which we were anxious not to alter, were not designed with a view to allowing a link-editing phase.

Available resources

The resources available at the outset of the project can be summarised under the main headings of equipment, programs and staff. (Symmetrically 'hardware', 'software' and worst of all, 'liveware'.)

For *hardware* to be available, it must be present, working, and accessible to those developing the system, and we have been fortunate in each respect. The system authors have gained frequent access to the hardware, largely by a high degree of willingness on the part of those running the normal production work of the machine to allow such access. This in turn was a consequence of the fact that the same person (M.W.) was responsible both for the software project and the production service. When questions of priority have arisen they have been resolved immediately; we feel that what matters is that the decision be immediate and binding rather than delayed in an attempt to ensure correctness.

**Those who visited Leeds prior to 1966 will recall that at that time the Computing Laboratory was housed in a nonconformist Chapel known as Eldon Hall.*

It is a pleasure to put on record our appreciation of the way in which Mr. H. Eastwood and the operators of KDF9 have co-operated on this matter.

The actual details of the hardware can be subdivided into three main types:

(i) Conventional, such as one finds on many medium sized machines. The relevant items here are the core store (32K × 48 bit words, ~6 μs cycle time), the disc store (4M words, 12K words/sec transfer rate, 100-300 ms access time), seven magnetic tape units and a sufficiency of conventional input/output peripherals.

(ii) Communications equipment. Basically this consists of a DEC PDP-8 computer, with 4K × 12 bit words, attached via a magnetic tape station buffer to the KDF9. The PDP-8 is equipped with the DEC 680 multiplexer subsystem, driving 32 teletypes, and was originally envisaged purely as a multiplexer to allow teletypes to communicate with KDF9. In fact the PDP-8 carries out a substantial amount of pre- and post-processing of messages within the system, and as such has provided a means of unifying many of the concepts.

(iii) Unconventional features of KDF9. In many respects the 'time-sharing' (i.e. multi-programming) version of KDF9 is unusual. KDF9 uses a system of push-down stores for arithmetic and subroutine organisation, and a set of 'Q-stores' for indexing.

On a multi-programming KDF9 the two sets of push-down stores and the Q-stores are quadruplicated; each set constitutes a 'level' of the machine, and has a priority associated with it. Privileged mode instructions allow processing to transfer between one level and another. In addition there is elaborate provision to allow user programs direct control of peripheral devices, rather than calling on the supervisory routine (known as 'director' on KDF9) to carry out peripheral transactions. This includes registers to indicate which devices are allocated to the level currently running, to indicate which peripheral was busy if a level refers to it and causes an interrupt, and a 'lock-out' store. This defines which areas of store are currently involved in peripheral transactions and causes an interrupt if a program refers to a locked out area. The presence of this special purpose equipment, much of it accessible when in non-privileged mode, has simplified the organisation of multi-programming within one level, as well as between several levels.

The *software* available on KDF9 at the time of its upgrading was frankly inadequate, both in concept and completeness. It operated at three levels of impact on the user; there was a supervisory routine (director), a file maintenance system (Prompt), and three compilers (an assembly language and two ALGOL compilers).

The director was responsible for peripheral allocation and for administering transfers on the shared peripherals (i.e. monitor flexowriter, disc store and the output well). The output well required manual intervention at any decision point, and there was no input well or job queue, the system relying on operators to insert jobs into the correct level by messages typed to the director. There was no provision for accounting information, other than flexowriter messages at job termination. However, the director coped well with some job mixes, and a sufficiently adroit team of operators could run the CPU about 50% of the available time.

The Prompt file maintenance system was a disc-based development of the earlier magnetic tape system, Post. In Post, the linear nature of magnetic tape had forced users to apply their corrections to a text within a single forward scan through the file using context seeking directives to control the amender. It was also necessary for the operators to separate the actions of Post into an 'assembly' stage of file creation and editing and a 'translation' stage which generated binary programs for subsequent execution; within each stage messages had to be applied in the order of appearance of texts on the tape. In Prompt, where files were held on the disc, it was no longer necessary to separate or sort messages. This was operationally more convenient, but was of no benefit to users. The amender was still based on a linear scan of the text, and the representation of text within the system was geared to this end.

The compilers available were for Usercode and ALGOL. Usercode is an assembly language with good mnemonics for operations, but very limited addressing techniques. For ALGOL two compatible compilers were available. The 'Whetstone' system (Randell and Russell, 1964) offers excellent diagnostic facilities; compilation is extremely fast, but the subsequent interpretation is slow. The 'Kidsgrove' system (Hawkins and Huxtable, 1963) is an optimising compiler, with long compilation times, and rather limited diagnostics, but producing a fairly efficient object code. No compiler allowed library insertion at compile time; instead each text file contained a copy of any library material, inserted when the file was created.

The *staff* available for the implementation of the project was limited, both in number and amount of time free for work on the project. There were four lecturers (M. Wells, D. Holdsworth, A. P. McCann, and D. H. Sleeman) and a rather variable number of computing assistants who are of assistant lecturer status. (A. A. Hock, D. J. Hainsworth, C. V. Harman, and D. K. Johnson.) All of these persons had many duties over and above their work on the development of the system.

Design considerations of the system

The main aims of the system have already been mentioned. In order to achieve these in a sensible time it was obvious that a prime consideration was the necessity to retain compatibility with existing software wherever possible. For our purposes we may think of various levels of compatibility:

level (i) 'language'—i.e. the same manuscript of data, program or system directives is acceptable to old and new system alike.

level (ii) 'representation'—i.e. the same deck of cards, paper tape or magnetic tape is acceptable to both systems.

level (iii) 'implementation'—i.e. the actual processing of input is achieved by the same coding in both systems.

In general, compatibility at one level implies compatibility at lower levels, and wherever possible level (iii) compatibility allows the use of existing code with the minimum of re-writing. We faced two major areas at which compatibility was impossible from the outset. At level (i) we were proposing a much more flexible system, needing a much more powerful command language to drive it. At level (ii) we were introducing a completely new peripheral unit, the teletype, with its limited character set, making it impossible to represent text as on cards or paper tape. The rather depressing conclusion at this stage is that compatibility at level (iii) is thus automatically denied. We shall see later that one of our implementation disciplines restored this compatibility.

A second design consideration, which influenced much of

our thinking, was the desire to minimise the permanent core store requirements of the system. The existing file maintenance systems (Post and Prompt) used large amounts of core (10·8K and 12·8K respectively) regardless of their current activity. Since we expected that most users would spend most time maintaining files, we could not take over the amenders from these systems. However, we could use the compilers. A compiler, though frequently treated as if it were part of an operating system, is not basically different from any other program. It reads in data, in this case the text of a program, and outputs results, in this case failure messages and/or a binary version of the program. Since we allowed directives to initiate the running of a users program, the same mechanism could serve to initiate the running of a compiler. In this way those activities of the system requiring large amounts of store can be run outside the area permanently allocated to the system; only those activities using a small amount of store are retained within the operating system's kernel. Within this area, the necessity of storing information describing the current activity of each on-line user meant that any process had to be implemented via a series of overlayed segments. Thus, the system quite naturally grew as a large number of independently written segments. For these to be produced, by different authors, we standardised the interfaces between segments. Each segment communicates with others either via the kernel of the operating system, for the transfer of control from one segment to the next, or by passing information across on the disc. It is natural here to use the same character set at all times, and we chose to use the set already used by Prompt (and Post). It was found possible to arrange that PDP-8 could perform the conversion from teletype code to this internal code, and vice versa (these conversions are not trivial). Since much of Prompt already used this internal character set, not only for file storage, but also for incoming and outgoing messages, we were able to regain compatibility at level (iii), and so make use of much existing software.

Within Prompt the basic unit of textual information is the file. A file is made up of a number of blocks, each of 640 words, the amount transferred in one disc revolution, and each block contains a pointer to the next block. The first block defines the owner of the file, the date it was last written, the file identifier, and the disc addresses of all blocks constituting this file, thus simplifying the returning of free space when a file is deleted. A block of a file contains a number of lines, each corresponding to a printed line, and made up of eight bit symbols packed six to a word. The first 'symbol' is in fact the length of the line in words, together with marker bits used for some special purpose lines. A symbol corresponds to a single character e.g. a or A or 7 or a group of characters, e.g. **begin** or :=. All communication within the Eldon 2 system is in terms of these symbols, either as structured lines, or as a simple unstructured stream of symbols. Obviously conversions between this device independent internal form and a device dependent external form will be performed by software. For on-line users this software is in the PDP-8; not only does this reduce the amount of store needed in a crucial area, but it also allows PDP-8 to filter out mistyped messages without interaction with KDF9. For other input media it is necessary only to provide a program to read messages and convert them to a stream of symbols on a magnetic tape. Means exist to allow such a tape to become a 'console' of the system. In this way a new medium can be interfaced in a (relatively) straightforward way.

When Eldon 2 is running with on-line consoles attached, the store is split into a number of areas as shown in **Fig. 1**.

The director program (3,200 words) organises multiprogramming among four hardware levels.* One of these,

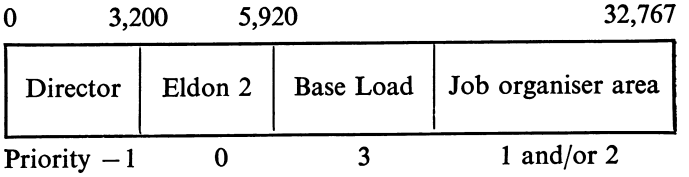

Fig. 1.

running with priority 0 and occupying 2,700 words controls the foreground system, i.e. activities which are initiated at consoles and returning results to consoles. This is shown as Eldon 2. A second level, with priority 3, holds a base load, a long running job with minimal peripheral activity. It is in this level that the CPU spends most of its time. The remaining two levels and residue of core store (shown as job organiser area) are used for short duration operator initiated jobs and job-queue servicing.

There are two main types of activity within the foreground system, the creation and maintenance of files, and the translation and execution of programs stored as files. Operations within the foreground system take place either within level 0, i.e. within Eldon 2, or by the creation of entries in a foreground job queue held on the disc, the corresponding job then being processed in level 1 or 2, where it multi-programs with other levels. Eldon 2 is itself a multi-programming environment, with a core map as shown in **Fig. 2.**

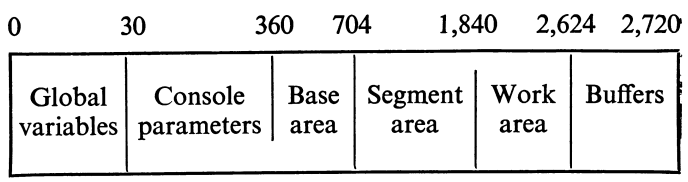

Fig. 2. The numbers give the addresses of the boundaries

In effect the global variables, console parameters, base area and buffers act as an executive, providing peripheral communication with the consoles and overlay control for the set of re-entrant segments. The global variables provide the interface between executive and the segments which are obeyed in the segment area and share a common work area. The executive allows the author of a segment to write as if he were communicating with a single console, supervising a separate process for each console, with at most one process using KDF9 at a given time. It contains a simple scheduling algorithm to decide which process to resume when the current one becomes held up as a result of console activity, by reinstatement of the relevant work area, and if necessary, the segment, which is then entered. All segments and work areas are buffered on the disc. If no process can be resumed executive forces an interrupt and director will enter another level. Thus for a job to be processed within Eldon 2 it must be capable of running in the area set aside for a segment and its work space. Any activity which cannot be handled in this way is dealt with by the 'job-queue' segment.

The job-queue segment, when called in, creates an entry in the foreground job-queue; this is held on the disc and

*These have priorities 0, 1, 2, 3; director has a notional priority of −1. At any time, the lowest numbered priority level that is free to run will do so.

contains space for one entry from each console. Having created the entry the job-queue segment informs director of the amount of store requested for this foreground job. If there is a free level (say level 1) director loads job-organiser into this level, with a store allocation at least as large as this request. If this amount of store is not available then the operator initiated level 2 job will be 'rolled out' onto the disc to free the required store. Experience has shown that the level 3 base load should not be rolled out in this way. This 'locking-in' of the level 3 job is an important factor in the achievement of high CPU utilisation, because the CPU always has some work to do. If no level is available to meet a foreground job request then the input of job-organiser has to wait until a level becomes free. This situation normally arises if a foreground job is already running. Because of the short time-limit (20 seconds) imposed on foreground jobs, the level soon becomes free. When job-organiser finds that the foreground job-queue is empty it reinstates a rolled out job. If none exists, then the background queue is serviced.

While the combined efforts of director and job-organiser arrange the running of a particular job the corresponding job-queue segment is resumed approximately every 30 seconds and inspects the job-queue entry to discover whether the job has been run. When this is so, the job-queue segment then transfers any output which the job has written to the disc back to the console which initiated the job. The job-queue segment also detects a request from a user-console to cancel an entry in the job-queue, and carries out this request. Jobs may also be initiated by operators, either by a direct request to the KDF9 director, or by writing entries to a background job-queue, using a job called in for this purpose. A user at a console may also create entries in the background job-queue. All entries in either job-queue are processed by the job-organiser. Whenever a KDF9 level terminates, or reduces its store requirement, the director consolidates the store. All the space not currently occupied by director or by a program is made contiguous, and director loads the job-organiser into this area and enters it as if it were a program. Whenever a job which corresponds to a foreground job-queue entry terminates, director marks the corresponding job-queue entry, to indicate that the job-queue segment may resume in order to return any output to the console.

When the on-line consoles are no longer required, Eldon 2 is terminated (a console command allows this), and the space it occupied is now available for normal use. The foreground queue will now be inactive but of course there may still be console initiated jobs in the background queue.

Current status

This system has been running for 18 months for 22½ hours a day 6 days a week; during the day the on-line system is in use from 9.30 a.m. to 11.00 p.m. Currently 32 teletypes are in regular use. With 32 consoles the CPU spends about 2% of its time in the Eldon 2 level, about 15% processing the foreground job-queue, about 65% processing the base load and 18% in director. When the consoles are not in use, the time spent in director drops to about 15%, and 85% is devoted to processing user jobs.

The languages available are Usercode and KAL4 (an improved assembly language with more powerful addressing facilities and a block structure), Whetstone and Kidsgrove ALGOL, and FORTRAN. This last was written in the Department in about six man months, and operates by compiling into KAL4, which is then assembled. The original intention was to process small jobs quickly, and for this purpose the compiler has also been made available as a stand alone in-core system operating direct from cards to the line printer (*Cf.* WATFOR). To our surprise, compilation speeds are in excess of 500 statements a minute, and the resulting object code seems to be better than that from the Kalgol compiler.

We anticipated that system overheads might reduce CPU utilisation, causing a short fall of *computing* capacity. In the event, utilisation has improved from ~45% to ~85%, and the major difficulty has been a short fall of file storage space. As a consequence of the ease of use, and rapid turn round that the system offers, users have spontaneously moved nearly all their work into this system, and we are now seriously embarrassed by lack of space on the disc. At present the files containing the text of the system itself amount to 190 K bytes (each byte representing a single basic symbol) for the foreground system and about 900 K bytes for the extra facilities of the background system. Users texts amount to about 15M bytes held on the disc and a further 40M bytes held on the system back up tapes. We have thus been forced to implement systematic archiving of unaccessed material and restoring of archived files, over and above the normal system of incremental dumping required for system security.

There are 250 authorised users of the system each known to the system by his 'log-in name'. It is also available to any casual user, by logging in with a name beginning with a full stop. This bypasses the check of the log-in name against that of authorised users, but carries a penalty, in that files owned by casual users are regarded as expendable. Despite this, many of our undergraduates make extensive casual use of the system in connection with their final year project work.

Conclusions

We have learned a great deal from this system. We believe that a small team, working towards deliberately modest aims, can produce in a realistic time an operating environment which will satisfy users needs. We believe that the members of the team must, from the outset, accept the discipline implied by working with agreed interfaces, and that this will only be possible if the interface arises as a natural by-product of the way the work is undertaken.

A problem still unsolved by us is the difficulty of allowing undisturbed access to the system for the system writers. Obviously, in the early stages when crashes are frequent, normal users do not want to use the system. As reliability improves, and crashes become less frequent and less disastrous, it is possible to allow development and use to proceed in parallel. However, there are occasions when the system must be developed, with all terminals live, but not being used (e.g. when working on the multiplexer). At such times, users are apt suddenly to appear, and cause all manner of frustration to the system programmer who wishes to make controlled observation of the system.

References

BURNS, D., HAWKINS, E. N., JUDD, D. R., and VENN, J. H. (1966). *The Computer Journal*, Vol. 8, No. 4, p. 297.

HAWKINS, E. N., and HUXTABLE, D. H. R. (1963). A multi-pass translation scheme for Algol 60, *Annual Review in Automatic Programming*, Vol. 3, p. 163, Pergammon Press.

RANDELL, B., and RUSSELL, L. J. (1964). *Algol 60 Implementation.* Academic Press.